EEL 5764 Graduate Computer Architecture

Chapter 4 - Multiprocessors and TLP

Ann Gordon-Ross
Electrical and Computer Engineering
University of Florida

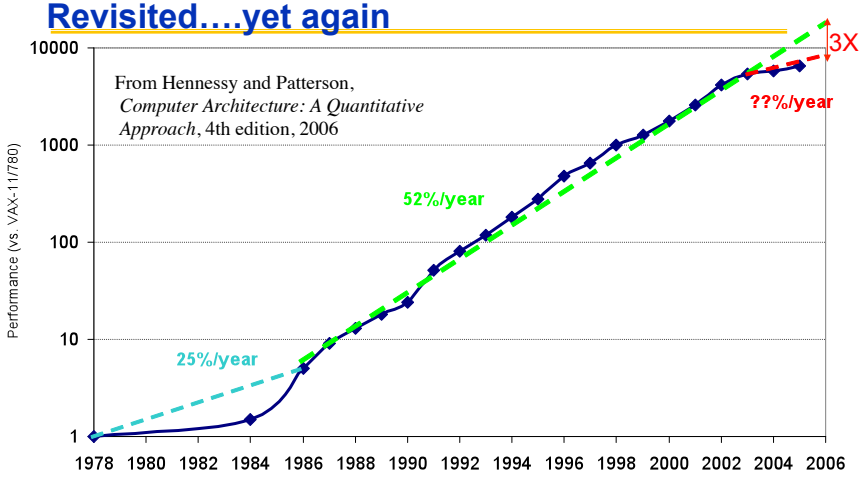
http://www.ann.ece.ufl.edu/

These slides are provided by:
David Patterson
Electrical Engineering and Computer Sciences, University of California, Berkeley
Modifications/additions have been made from the originals

Outline

- MP Motivation
- SISD v. SIMD v. MIMD
- Centralized vs. Distributed Memory
- Challenges to Parallel Programming
- Consistency, Coherency, Write Serialization
- Snoopy Cache
- Directory-based protocols and examples

Uniprocessor Performance (SPECint) - Revisited….yet again



- VAX : 25%/year 1978 to 1986
- RISC + x86: 52%/year 1986 to 2002
- RISC + x86: ??%/year 2002 to present

Déjà vu all over again?

"… today's processors … are nearing an impasse as technologies approach the speed of light.."

David Mitchell, *The Transputer: The Time Is Now* (1989)

- Transputer had bad timing (Uniprocessor performance↑)
  ⇒ Procrastination rewarded: 2X seq. perf. / 1.5 years
- "We are dedicating all of our future product development to multicore designs. … This is a sea change in computing"

  Paul Otellini, President, Intel (2005)
- All microprocessor companies switch to MP (2X CPUs / 2 yrs)
  ⇒ Procrastination penalized: 2X sequential perf. / 5 yrs

Manufacturer/Year	AMD/'05	Intel/'06	IBM/'04	Sun/'05
Processors/chip	2	2	2	8
Threads/Processor	1	2	2	4
Threads/chip	2	4	4	32

Other Factors Pushing Multiprocessors

- Growth in data-intensive applications
  - Data bases, file servers, …
  - Inherently parallel - SMT can't fully exploit
- Growing interest in servers, server perf.
  - Internet
- Increasing desktop perf. less important (outside of graphics)
  - Don't need to run Word any faster
  - But near unbounded performance increase has lead to terrible programming

Other Factors Pushing Multiprocessors

- Lessons learned:
  - Improved understanding in how to use multiprocessors effectively
    - » Especially in servers where significant natural TLP
  - Advantages in replication rather than unique design
    - » In uniprocessor, redesign every few years = tremendous R&D
      - Or many designs for different customer demands (Celeron vs. Pentium)
    - » Shift efforts to multiprocessor
      - Simple add more processors for more performance

Outline

- MP Motivation
- SISD v. SIMD v. MIMD
- Centralized vs. Distributed Memory
- Challenges to Parallel Programming
- Consistency, Coherency, Write Serialization
- Snoopy Cache
- Directory-based protocols and examples

Flynn's Taxonomy

M.J. Flynn, "Very High-Speed Computers", *Proc. of the IEEE*, V 54, 1900-1909, Dec. 1966.

- Flynn divided the world into two streams in 1966 = instruction and data

Single Instruction Single Data (SISD) (Uniprocessor)	Single Instruction Multiple Data SIMD (single PC: Vector, CM-2)
Multiple Instruction Single Data (MISD) (????)	Multiple Instruction Multiple Data MIMD (Clusters, SMP servers)

- SIMD ⇒ Data Level Parallelism
- MIMD ⇒ Thread Level Parallelism
- MIMD popular because
  - Flexible: N pgms and 1 multithreaded pgm
  - Cost-effective: same MPU in desktop & MIMD

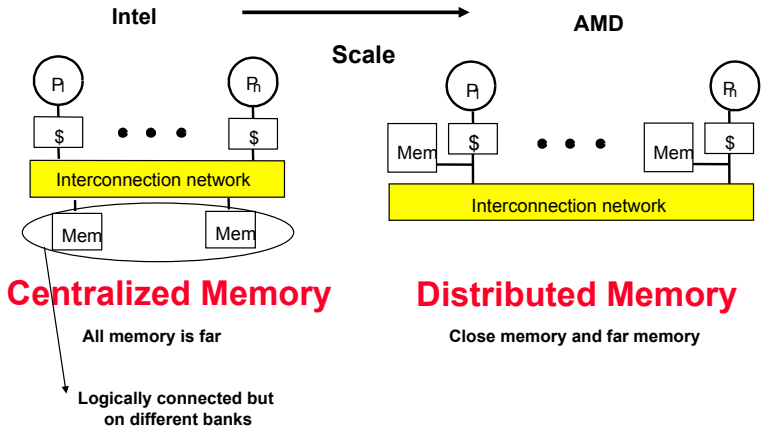
Outline

- MP Motivation
- SISD v. SIMD v. MIMD
- Centralized vs. Distributed Memory
- Challenges to Parallel Programming
- Consistency, Coherency, Write Serialization
- Snoopy Cache
- Directory-based protocols and examples

Back to Basics

- A parallel computer is…
  - … a collection of processing elements that cooperate and communicate to solve large problems fast.
- How do we build a parallel architecture?
  - Computer Architecture + Communication Architecture
- 2 classes of multiprocessors WRT memory:
  1. Centralized Memory Multiprocessor
     - Take a single design and just keep adding more processors/cores
       - few dozen processor chips (and < 100 cores) in 2006
     - Small enough to share single, centralized memory
     - But interconnect is becoming a bottleneck…..
  2. Physically Distributed-Memory multiprocessor
     - Can have larger number chips and cores
     - BW demands are met by distributing memory among processors

Centralized vs. Distributed Memory



Centralized Memory Multiprocessor

- Also called symmetric multiprocessors (SMPs)
  - main memory has a symmetric relationship to all processors
    - All processors see same access time to memory
- Reducing interconnect bottleneck
  - Large caches ⇒ single memory can satisfy memory demands of small number of processors
- How big can the design realistically be?
  - Scale to a few dozen processors by using a switch and by using many memory banks
  - Scaling beyond that is technically conceivable but….it becomes less attractive as the number of processors sharing centralized memory increases
    - Longer wires = longer latency
    - Higher load = higher power
    - More contention = bottleneck for shared resource

Distributed Memory Multiprocessor

- Distributed memory is a “must have” for big designs
- Pros:
  - Cost-effective way to scale memory bandwidth
    - If most accesses are to local memory
  - Reduces latency of local memory accesses
- Cons:
  - Communicating data between processors more complex
  - Software aware
    - Must change software to take advantage of increased memory BW

2 Models for Communication and Memory Architecture

1. message-passing multiprocessors
   - Communication occurs by explicitly passing messages among the processors
2. shared memory multiprocessors
   - Communication occurs through a shared address space (via loads and stores):
     either
     - UMA (Uniform Memory Access time) for shared address, centralized memory MP
     - NUMA (Non Uniform Memory Access time multiprocessor) for shared address, distributed memory MP
       - More complicated
- In past, confusion whether “sharing” means sharing physical memory (Symmetric MP) or sharing address space

Outline

- MP Motivation
- SISD v. SIMD v. MIMD
- Centralized vs. Distributed Memory
- Challenges to Parallel Programming
- Consistency, Coherency, Write Serialization
- Snoopy Cache
- Directory-based protocols and examples

Challenges of Parallel Processing

- First challenge is % of program inherently sequential
- Suppose 80X speedup from 100 processors. What fraction of original program can be sequential?

  a. 10%

  b. 5%

  c. 1%

  d. <1%

Amdahl's Law Answers

$$\text{Speedup}_{\text{overall}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{parallel}}}{\text{Speedup}_{\text{parallel}}}}$$

$$80 = \frac{1}{(1 - \text{Fraction}_{\text{parallel}}) + \frac{\text{Fraction}_{\text{parallel}}}{100}}$$

$$80 \times ((1 - \text{Fraction}_{\text{parallel}}) + \frac{\text{Fraction}_{\text{parallel}}}{100}) = 1$$

$$79 = 80 \times \text{Fraction}_{\text{parallel}} - 0.8 \times \text{Fraction}_{\text{parallel}}$$

$$\text{Fraction}_{\text{parallel}} = 79 / 79.2 = 99.75\%$$

Challenges of Parallel Processing

- Second challenge is long latency to remote memory
- Suppose 32 CPU MP, 2GHz, 200 ns remote memory (400 clock cycles), all local accesses hit memory hierarchy and base CPI is 0.5.
- What is the performance impact if 0.2% instructions involve remote access?

  a. 1.5X

  b. 2.0X

  c. 2.5X

CPI Equation

- CPI = Base CPI +
  Remote request rate
  x Remote request cost
- CPI = 0.5 + 0.2% x 400 = 0.5 + 0.8 = 1.3
- No communication is 1.3/0.5 or 2.6 faster than when 0.2% instructions involve remote access

Challenges of Parallel Processing

1. Need new advances in algorithms
   - Application parallelism
2. New programming languages
   - Hard to program parallel applications
3. How to deal with long remote latency impact
   - both by architect and by the programmer
   - For example, reduce frequency of remote accesses either by
     - » Caching shared data (HW)
     - » Restructuring the data layout to make more accesses local (SW)

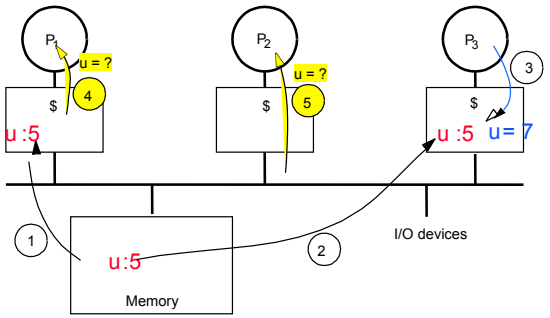
Outline

- MP Motivation
- SISD v. SIMD v. MIMD
- Centralized vs. Distributed Memory
- Challenges to Parallel Programming
- Consistency, Coherency, Write Serialization
- Snoopy Cache
- Directory-based protocols and examples

Symmetric Shared-Memory Architectures - UMA

- From multiple boards on a shared bus to multiple processors inside a single chip
- Equal access time for all processors to memory via shared bus
- Each processor will cache both
  - Private data are used by a single processor
  - Shared data are used by multiple processors
- Advantage of caching shared data
  - Reduces latency to shared data, memory bandwidth for shared data, and interconnect bandwidth
  - But adds cache coherence problem

Example Cache Coherence Problem



- Processors see different values for u after event 3
- With write back caches, depends on which cache flushes first
  - » Processes accessing main memory may see very stale value
- Unacceptable for programming, and its frequent!

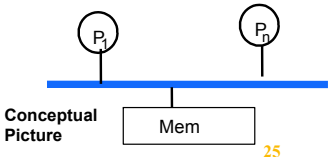
Not Just Cache Coherency….

- Getting single variable values coherent isn't the only issue
  - Coherency alone doesn't lead to correct program execution
- Also deals with synchronization of different variables that interact
  - Shared data values not only need to be coherent but order of access to those values must be protected

Example

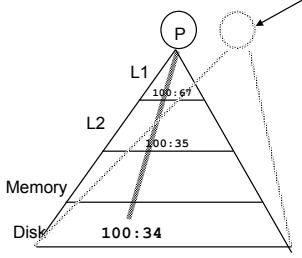
P_1	P_2
/*Assume initial value of A and flag is 0*/	
`A = 1;`	`while (flag == 0);` /*spin idly*/
`flag = 1;`	`print A;`

- expect memory to respect order between accesses to *different* locations issued by a given process
  - to preserve orders among accesses to same location by different processes
- Coherence is not enough!
  - pertains only to single location



Intuitive Memory Model

This process *should* see value written immediately



- Reading an address should return the last value written to that address
  - Easy in uniprocessors
  - In multiprocessors, more complicated than just seeing the last value written
    - » How do you define write order between different processes
- Too vague and simplistic; 2 issues
  1. Coherence defines values returned by a read
  2. Consistency determines when a written value will be returned by a read
- Coherence defines behavior to same location, Consistency defines behavior to other locations

Defining Coherent Memory System

1. Preserve Program Order: A read by processor P to location X that follows a write by P to X, with no writes of X by another processor occurring between the write and the read by P, always returns the value written by P
2. Coherent view of memory: Read by a processor to location X that follows a write by another processor to X returns the written value if the read and write are *sufficiently separated in time* (hardware recognition time) and no other writes to X occur between the two accesses
3. Write serialization: 2 writes to same location by any 2 processors are seen in the same order by all processors
   - If not, a processor could keep value 1 since saw as last write
   - For example, if the values 1 and then 2 are written to a location, processors can never read the value of the location as 2 and then later read it as 1

Write Consistency

- For now assume

1. A write does not complete (and allow the next write to occur) until all processors have seen the effect of that write
2. The processor does not change the order of any write with respect to any other memory access

⇒ if a processor writes location A followed by location B, any processor that sees the new value of B must also see the new value of A

- These restrictions allow the processor to reorder reads, but forces the processor to finish writes in program order

Outline

- MP Motivation
- SISD v. SIMD v. MIMD
- Centralized vs. Distributed Memory
- Challenges to Parallel Programming
- Consistency, Coherency, Write Serialization
- Snoopy Cache
- Directory-based protocols and examples

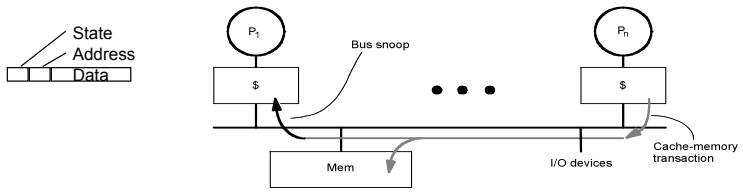
Basic Schemes for Enforcing Coherence

- Problem = Program on multiple processors will normally have copies of the same data in several caches
- Rather than trying to avoid sharing in SW, SMPs use a HW protocol to maintain coherent caches through:
  - Migration - data can be moved to a local cache and used there in a transparent fashion
    - » Reduces both latency to access shared data that is allocated remotely and bandwidth demand on the shared memory
  - Replication – for shared data being simultaneously read, since caches make a copy of data in local cache
    - » Reduces both latency of access and contention for read shared data

2 Classes of Cache Coherence Protocols

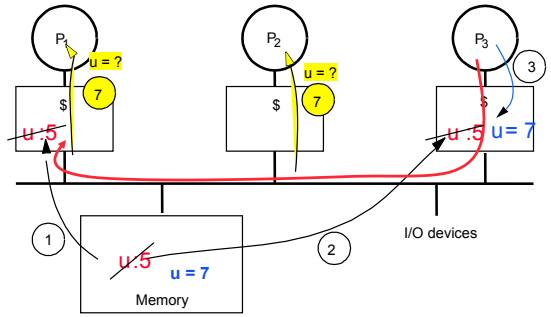
1. Snooping — Every cache with a copy of data also has a copy of sharing status of block, but no centralized state is kept
   - All caches are accessible via some broadcast medium (a bus or switch)
   - All cache controllers monitor or snoop on the medium to determine whether or not they have a copy of a block that is requested on a bus or switch access
   - Emphasis for now with systems because they are small enough
2. Directory based — Sharing status of a block of physical memory is kept in just one location, the directory
   - Old method revisited to deal with future larger systems
   - Moving from bus topology to switch topology

Snooping Cache-Coherence Protocols



- Each processors cache controller "snoops" all transactions on the shared medium (bus or switch)
  - Attractive solution with common broadcast bus
  - Only interested in relevant transaction
  - take action to ensure coherence
    - » invalidate, update, or supply value
  - depends on state of the block and the protocol
- Either get exclusive access before write via write invalidate or update all copies on write
- Advantages:
  - Distributed model
  - Only a slightly more complicated state machine
  - Doesn't cost much WRT hw

Example: Write-thru Invalidate



- Must invalidate before step 3
- Could just broadcast new data value, all caches update to reflect
  - Write update uses more bandwidth - too much
  - all recent MPUs use write invalidate

Architectural Building Blocks - What do we need?

- Cache block state transition diagram
  - FSM specifying how state of block changes
    - » invalid, valid, dirty
  - Logically need FSM for each cache block, not how it is implemented but we will envision this scenario
- Broadcast Medium (e.g., bus)
  - Logically single set of wires connect several devices
  - Protocol: arbitration, command/addr, data
  - ⇒ Every device observes every transaction
- Broadcast medium enforces serialization of read or write accesses ⇒ Write serialization
  - 1st processor to get medium invalidates others copies
  - Implies cannot complete write until it obtains bus
- Also need method to find up-to-date copy of cache block
  - If write-back, copy may be in anther processors L1 cache

How to locate up-to-date copy of data

- **Write-through:**
  - Reads always get up-to-date copy from memory
  - Write through simpler if enough memory BW
- **Write-back harder**
  - Most recent copy can be in any cache
  - Lower memory bandwidth
  - Most multiprocessors use write-back
- **Can use same snooping mechanism**
  1. Snoop every address placed on the bus
  2. If a processor has dirty copy of requested cache block, it provides it in response to a read request and aborts the memory access
  - Complexity from retrieving cache block from a processor cache, which can take longer than retrieving it from memory (which is optimized)

Cache Resources for WB Snooping

- Normal cache tags can be used for snooping
- Valid bit per block makes invalidation easy
- Reads
  - misses easy since rely on snooping
  - Processors respond if they have dirty data from a read miss
- Writes
  - Need to know whether any other copies of the block are cached
    - » No other copies ⇒ No need to place write on bus for WB
    - » Other copies ⇒ Need to place invalidate on bus

Cache Resources for WB Snooping

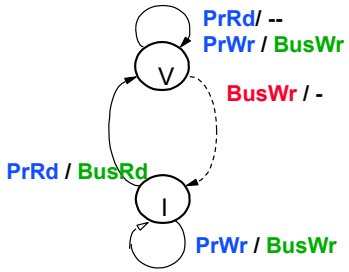
- Need one extra state bit to track whether a cache block is shared
  - Write to Shared block ⇒ Need to place invalidate on bus and mark cache block as exclusive (if an option)
  - No further invalidations will be sent for that block
  - This processor called owner of cache block
  - Owner then changes state from shared to unshared (or exclusive)

Example Protocol - Start Simple

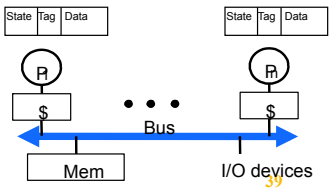
- Snooping coherence protocol is usually implemented by incorporating a finite-state controller in each node
- Logically, think of a separate controller associated with each cache block
  - That is, snooping operations or cache requests for different blocks can proceed independently
- In implementations, a single controller allows multiple operations to distinct blocks to proceed in interleaved fashion
  - that is, one operation may be initiated before another is completed, even through only one cache access or one bus access is allowed at time

Write-through Invalidate Protocol

- 2 states per block in each cache
  - as in uniprocessor
  - Hardware state bits associated with blocks that are in the cache
  - other blocks can be seen as being in invalid (not-present) state in that cache
- Writes invalidate all other cache copies (write no-alloc)
  - can have multiple simultaneous readers of block,but write invalidates them



PrRd: Processor Read
PrWr: Processor Write
BusRd: Bus Read
BusWr: Bus Write

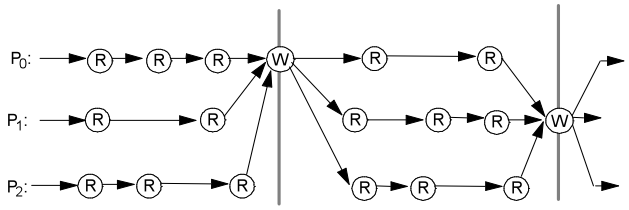


Is 2-state Protocol Coherent?

- Processor only observes state of memory system by issuing memory operations
  - If processor only does ALU operations, doesn't see see state of memory
- Assume bus transactions and memory operations are atomic and a one-level cache
  - one bus transaction complete before next one starts
  - processor waits for memory operation to complete before issuing next
  - with one-level cache, assume invalidations applied during bus transaction
- All writes go to bus + atomicity
  - Writes serialized by order in which they appear on bus (bus order)

  => invalidations applied to caches in bus order
- How to insert reads in this order?
  - Important since processors see writes through reads, so determines whether write serialization is satisfied
  - But read hits may happen independently and do not appear on bus or enter directly in bus order
- Let's understand other ordering issues

Ordering



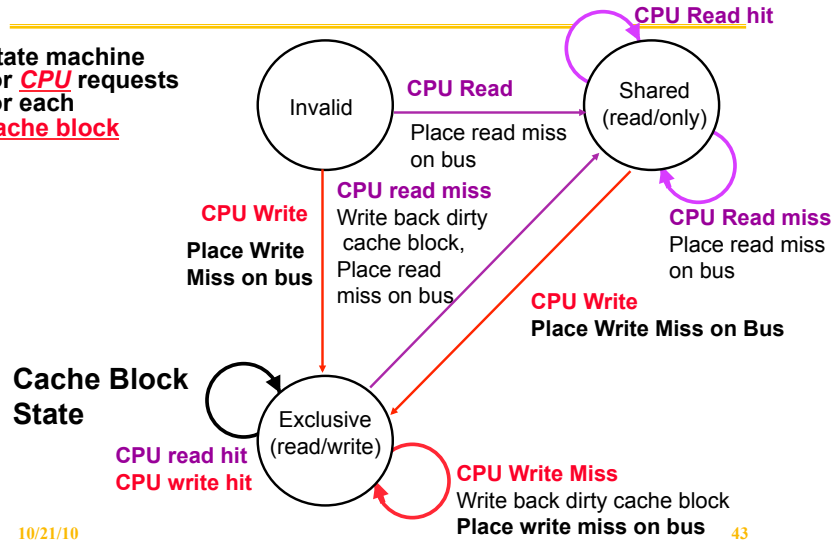
- Writes establish a partial ordering for the reads
- Doesn't constrain ordering of reads, though shared-medium (bus) will order read misses too
  - any order among reads between writes is fine, as long as in program order

Example Write Back Snoopy Protocol

- Look at invalidation protocol with a write-back cache
  - Snoops every address on bus
  - If cache has a dirty copy of requested block, provides that block in response to the read request and aborts the memory access
- Each memory block is in one state (implied):
  - Clean in all caches and up-to-date in memory (Shared)
  - OR Dirty in exactly one cache (Exclusive)
  - OR Not in any caches
- Each cache block is in one state (track these):
  - Shared : block can be read
  - OR Exclusive : cache has only copy, its writeable, and dirty
  - OR Invalid : block contains no data (in uniprocessor cache too)
- Read misses: cause all caches to snoop bus
- Writes to clean blocks are treated as misses
- Assume write-allocate in this example

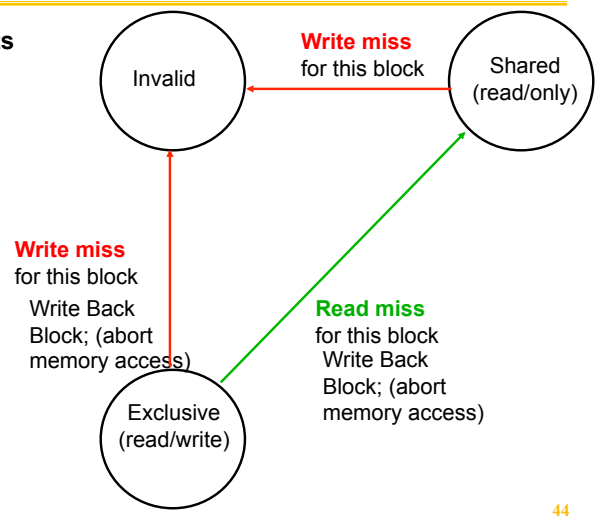
Write-Back State Machine - CPU

- State machine for *CPU* requests for each cache block



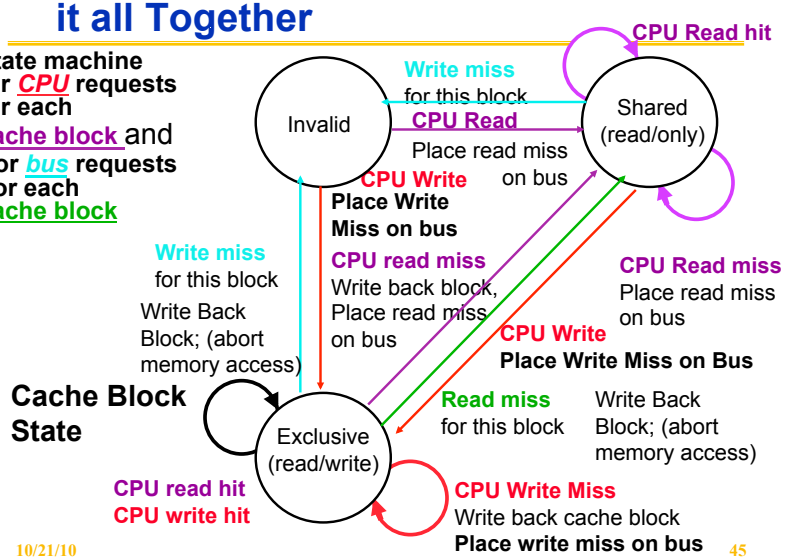
Write-Back State Machine- Bus request

- State machine for *bus* requests for each cache block



Write-back State Machine - Putting it all Together

- State machine for *CPU* requests for each cache block and for *bus* requests for each cache block



Example

	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1 Write 10 to A1												
P1: Read A1												
P2: Read A1												
P2: Write 20 to A1												
P2: Write 40 to A2												

- Assumes A1 and A2 *map* to same cache location but are not in the same memory block (so not in the same cache block)
* Initial cache state is invalid
* Assume write allocate

Example

	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1 Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1												
P2: Read A1												
P2: Write 20 to A1												
P2: Write 40 to A2												

- Assumes A1 and A2 *map* to same cache location but are not in the same memory block (so not in the same cache block)
* Initial cache state is invalid
* Assume write allocate

Example

	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1 Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1												
P2: Write 20 to A1												
P2: Write 40 to A2												

- Assumes A1 and A2 *map* to same cache location but are not in the same memory block (so not in the same cache block)
* Initial cache state is invalid
* Assume write allocate

Example

Goes to shared because it is clean

	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1 Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1				Shar.	A1		RdMs	P2	A1			
	Shar.	A1	10				WrBk	P1	A1	10	A1	10
				Shar.	A1	10					A1	10
P2: Write 20 to A1												
P2: Write 40 to A2												

•Assumes A1 and A2 *map* to same cache location but are not in the same memory block (so not in the same cache block)
* Initial cache state is invalid
* Assume write allocate

Example

	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1 Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1				Shar.	A1		RdMs	P2	A1			
	Shar.	A1	10				WrBk	P1	A1	10	A1	10
				Shar.	A1	10					A1	10
P2: Write 20 to A1	Inv.			Excl.	A1	20	WrMs	P2	A1		A1	10
P2: Write 40 to A2												

•Assumes A1 and A2 *map* to same cache location but are not in the same memory block (so not in the same cache block)
* Initial cache state is invalid
* Assume write allocate

Example

	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1 Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1				Shar.	A1		RdMs	P2	A1			
	Shar.	A1	10				WrBk	P1	A1	10	A1	10
				Shar.	A1	10	RdDa	P2	A1	10	A1	10
P2: Write 20 to A1	Inv.			Excl.	A1	20	WrMs	P2	A1		A1	10
P2: Write 40 to A2							WrMs	P2	A2			
				Excl.	A2	40	WrBk	P2	A1	20	A1	20

•Assumes A1 and A2 *map* to same cache location but are not in the same memory block (so not in the same cache block)
* Initial cache state is invalid
* Assume write allocate

Implementation Complications

- **Write Races - Who writes first??**
  - **Cannot update cache until bus is obtained**
    - » **Otherwise, another processor may get bus first, and then write the same cache block!**
  - **Two step process:**
    - » **Arbitrate for bus**
    - » **Place miss on bus and complete operation (update cache)**
  - **If write miss occurs to block while waiting for bus, handle miss (invalidate may be needed) and then restart.**
  - **Split transaction bus:**
    - » **Bus transaction is not really atomic: can have multiple outstanding transactions for a block**
    - » **Multiple misses can interleave, allowing two caches to grab block in the Exclusive state**
    - » **Must track and prevent multiple misses for one block**
- **Must support interventions and invalidations**

Limitations in Symmetric Shared-Memory Multiprocessors and Snooping Protocols

- Single memory accommodate all CPUs even though there may be multiple memory banks
- Bus-based
  - must support both coherence traffic & normal memory traffic
  - Solution:
    - » Multiple buses or interconnection networks (cross bar or small point-to-point)

Performance of Symmetric Shared-Memory Multiprocessors

- Cache performance is combination of
  1. Uniprocessor cache miss traffic
  2. Traffic caused by communication
     - » Results in invalidations and subsequent cache misses
- 4th C: *coherence miss*
  - Joins Compulsory, Capacity, Conflict
  - How significant are coherence misses?

Coherency Misses

1. True sharing misses
   - Processes must share data for communication or processing
   - Types:
     - Invalidates due to 1st write to shared block
     - Reads by another CPU of modified block in different cache
     - Miss would still occur if block size were 1 word
2. False sharing misses
   - When a block is invalidated because some word in the block, other than the one being read, is written into
   - Invalidation does not cause a new value to be communicated, but only causes an extra cache miss
   - Block is shared, but no word in block is actually shared ⇒ miss would not occur if block size were 1 word
   - Larger block sizes lead to more false sharing misses

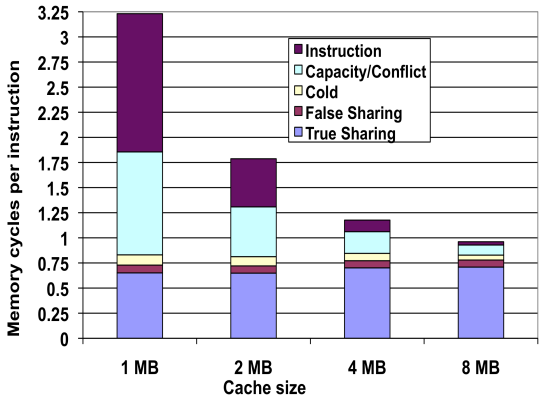
Example: True v. False Sharing v. Hit?

- Assume x1 and x2 in same cache block, different addresses in that block.
  P1 and P2 both read x1 and x2 before.

Time	P1	P2	True, False, Hit? Why?
1	Write x1		Hit, invalidate x1/x2 in P2
2		Read x2	False miss; x1 irrelevant to P2
3	Write x1		Hit, invalidate x1/x2 in P2
4		Write x2	False miss; x1 irrelevant to P2
5	Read x2		True miss

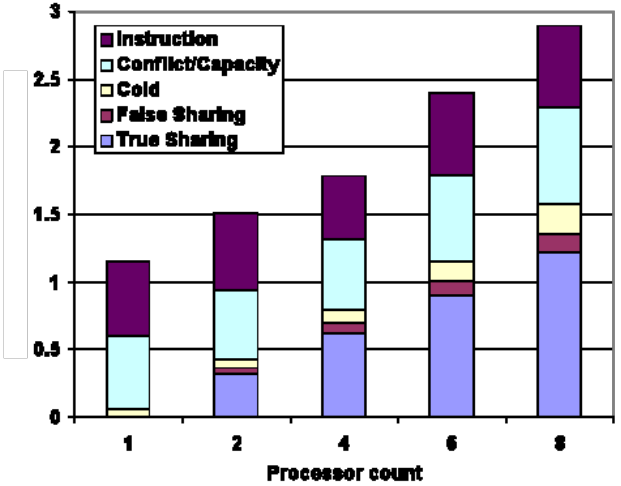
MP Performance 4 Processor Commercial Workload: OLTP, Decision Support (Database), Search Engine

True and false sharing doesn't change much as cache size increases



MP Performance 2MB Cache Commercial Workload: OLTP, Decision Support (Database), Search Engine

True and false sharing increase as number of CPUs increase. This will become more significant in the future as we move to many more processors



Outline

- Coherence
- Write Consistency
- Snooping
- Building Blocks
- Snooping protocols and examples
- Coherence traffic and Performance on MP
- Directory-based protocols and examples

A Cache Coherent System Must:

- Provide set of states, state transition diagram, and actions
- Manage coherence protocol
  - (0) Determine when to invoke coherence protocol
  - (a) Find info about state of block in other caches to determine action
    - » whether need to communicate with other cached copies
  - (b) Locate the other copies
  - (c) Communicate with those copies (invalidate/update)
- (0) is done the same way on all systems
  - state of the line is maintained in the cache
  - protocol is invoked if an "access fault" occurs on the line
- Different approaches (snoopy and directory based) distinguished by (a) to (c)

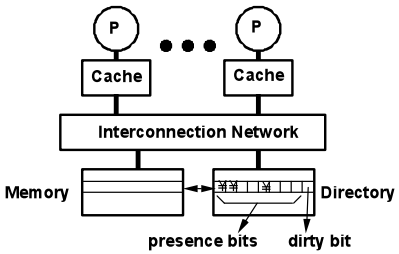
Bus-based Coherence

- All of (a), (b), (c) done through broadcast on bus
  - faulting processor sends out a "search"
  - others respond to the search probe and take necessary action
- Conceptually simple, but broadcast doesn't scale with p
  - on bus, bus bandwidth doesn't scale
  - on scalable network, every fault leads to at least p network transactions
- Scalable coherence, how do we keep track as the number of processors gets larger
  - can have same cache states and state transition diagram
  - different mechanisms to manage protocol - directory based

Scalable Approach: Directories

- Every memory block has associated directory information
  - keeps track of copies of cached blocks and their states
  - on a miss, find directory entry, look it up, and communicate only with the nodes that have copies if necessary
    - » Presence bit keeps track of which processors have it. Use bit vector to save space
    - » Minimizes traffic, don't just broadcast for each access
    - » Minimizes processing, not all processors have to check every address
  - in scalable networks, communication with directory and copies is through network transactions
- Many alternatives for organizing directory information

Basic Operation of Directory



- k processors.
- With each cache-block in memory: k presence-bits, 1 dirty-bit
- With each cache-block in cache: 1 valid bit, and 1 dirty (owner) bit

- Example:
  - Read from main memory by processor i:
    - » If dirty-bit OFF then { read from main memory; turn p[i] ON; }
    - » if dirty-bit ON then { recall line from dirty proc; update memory; turn dirty-bit OFF; turn p[i] ON; supply recalled data to i}
  - Write to main memory by processor i:
    - » If dirty-bit OFF then {send invalidations to all caches that have the block; turn dirty-bit ON; turn p[i] ON; ... }

Directory Protocol

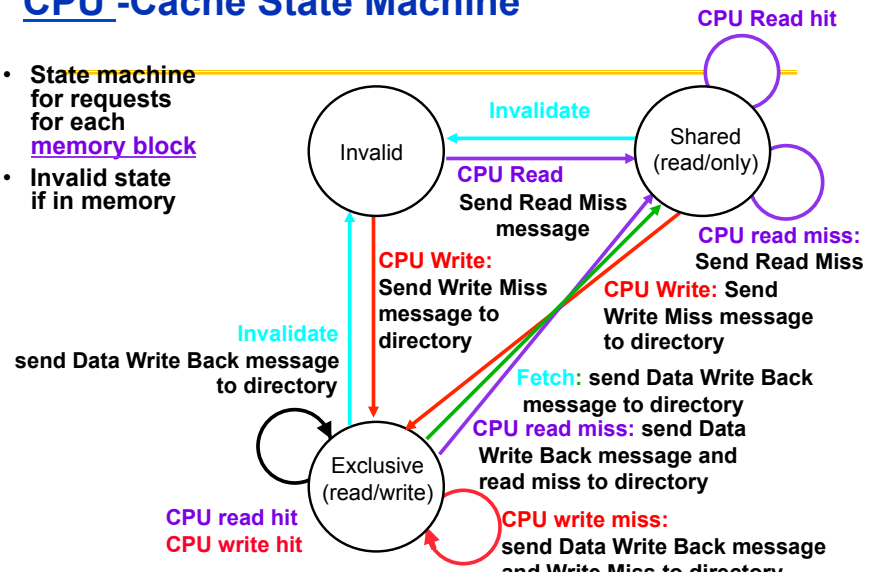
- Similar to Snoopy Protocol: Three states similar to snoopy
  - Shared: ≥ 1 processors have data, memory up-to-date
  - Uncached (no processor has it; not valid in any cache)
  - Exclusive: 1 processor (owner) has data; memory out-of-date
- In addition to cache state, must track which processors have data when in the shared state (usually bit vector, 1 if processor has copy) - presence vector
- Keep it simple:
  - Writes to non-exclusive data => write miss
  - Processor blocks until access completes
  - Assume messages received and acted upon in order sent (not realistic but we will assume)

State Transition Diagram for One Cache Block in Directory Based System

- States identical to snoopy case; transactions very similar.
- Transitions caused by read misses, write misses, invalidates, data fetch requests

CPU -Cache State Machine

- State machine for requests for each memory block
- Invalid state if in memory

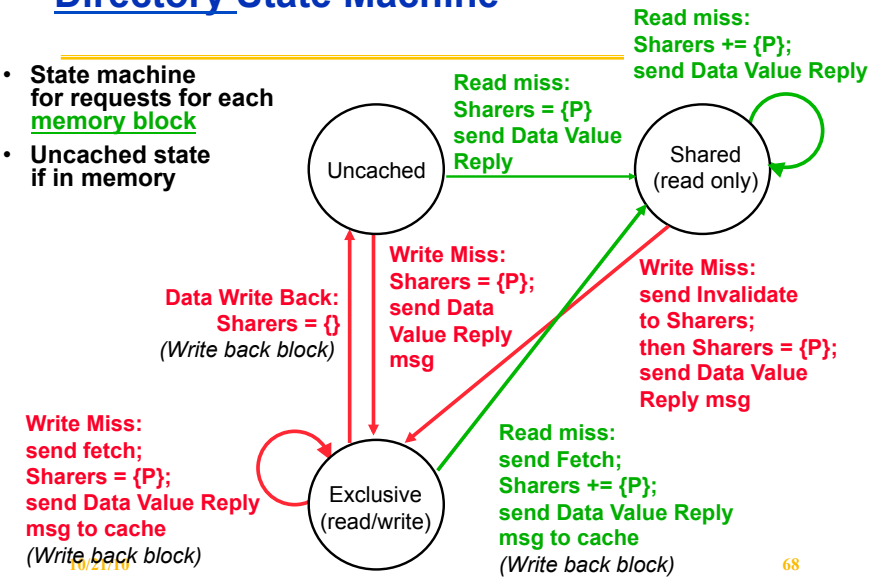


State Transition Diagram for Directory

- Same states & structure as the transition diagram for an individual cache
- 2 actions: update of directory state & send messages to satisfy requests
- Tracks all copies of memory block
- Also indicates an action that updates the sharing set, Sharers, as well as sending a message

Directory State Machine

- State machine for requests for each memory block
- Uncached state if in memory



Example

	Processor 1			Processor 2			Interconnect				Directory			Memory
	P1			P2			Bus				Directory			Memor
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	State	{Procs	Value
P1: Write 10 to A1														
P1: Read A1														
P2: Read A1														
P2: Write 20 to A1														
P2: Write 40 to A2														

•Assumes A1 and A2 *map* to same cache location but are not in the same memory block (so not in the same cache block)

* Initial cache state is invalid
* Assume write allocate

Example

	Processor 1			Processor 2			Interconnect				Directory			Memory
	P1			P2			Bus				Directory			Memor
step	State	Addr	Value	State	Addr	Value	Actio	Proc	Addr	Value	Addr	State	{Procs	Value
P1: Write 10 to A1							WrMs	P1	A1		A1	Ex	{P1}	
	Excl.	A1	10				DaRp	P1	A1	0				
P1: Read A1														
P2: Read A1														
P2: Write 20 to A1														
P2: Write 40 to A2														

•Assumes A1 and A2 *map* to same cache location but are not in the same memory block (so not in the same cache block)

* Initial cache state is invalid
* Assume write allocate

Example

	Processor 1			Processor 2			Interconnect				Directory			Memory
	P1			P2			Bus				Directory			Memor
step	State	Addr	Value	State	Addr	Value	Actio	Proc	Addr	Value	Addr	State	{Procs	Value
P1: Write 10 to A1							WrMs	P1	A1		A1	Ex	{P1}	
	Excl.	A1	10				DaRp	P1	A1	0				
P1: Read A1	Excl.	A1	10											
P2: Read A1														
P2: Write 20 to A1														
P2: Write 40 to A2														

•Assumes A1 and A2 *map* to same cache location but are not in the same memory block (so not in the same cache block)

* Initial cache state is invalid
* Assume write allocate

Example

	Processor 1			Processor 2			Interconnect				Directory			Memory
	P1			P2			Bus				Directory			Memor
step	State	Addr	Value	State	Addr	Value	Actio	Proc	Addr	Value	Addr	State	{Procs	Value
P1: Write 10 to A1							WrMs	P1	A1		A1	Ex	{P1}	
	Excl.	A1	10				DaRp	P1	A1	0				
P1: Read A1	Excl.	A1	10											
P2: Read A1				Shar.	A1		RdMs	P2	A1					
	Shar.	A1	10				Ftch	P1	A1	10	A1		{P1}	10
				Shar.	A1	10	DaRp	P2	A1	10	A1	Shar.	P1,P2}	10
P2: Write 20 to A1														
P2: Write 40 to A2														

Write Back

•Assumes A1 and A2 *map* to same cache location but are not in the same memory block (so not in the same cache block)

* Initial cache state is invalid
* Assume write allocate

Example

	Processor 1			Processor 2			Interconnect				Directory			Memory
	P1			P2			Bus				Directory			Memor
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	State	{Procs	Value
P1: Write 10 to A1							WrMs	P1	A1		A1	Ex	{P1}	
	Excl.	A1	10				DaRp	P1	A1	0				
P1: Read A1	Excl.	A1	10											
P2: Read A1				Shar.	A1		RdMs	P2	A1					
	Shar.	A1	10				Ftch	P1	A1	10	A1		{P1}	10
				Shar.	A1	10	DaRp	P2	A1	10	A1	Shar.	P1,P2}	10
P2: Write 20 to A1				Excl.	A1	20	WrMs	P2	A1					10
	Inv.						Inval.	P1	A1		A1	Excl.	{P2}	10
P2: Write 40 to A2														

•Assumes A1 and A2 *map* to same cache location but are not in the same memory block (so not in the same cache block)

* Initial cache state is invalid

* Assume write allocate

Example

	Processor 1			Processor 2			Interconnect				Directory			Memory
	P1			P2			Bus				Directory			Memor
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	State	{Procs	Value
P1: Write 10 to A1							WrMs	P1	A1		A1	Ex	{P1}	
	Excl.	A1	10				DaRp	P1	A1	0				
P1: Read A1	Excl.	A1	10											
P2: Read A1				Shar.	A1		RdMs	P2	A1					
	Shar.	A1	10				Ftch	P1	A1	10	A1		{P1}	10
				Shar.	A1	10	DaRp	P2	A1	10	A1	Shar.	P1,P2}	10
P2: Write 20 to A1				Excl.	A1	20	WrMs	P2	A1					10
	Inv.						Inval.	P1	A1		A1	Excl.	{P2}	10
P2: Write 40 to A2							WrMs	P2	A2		A2	Excl.	{P2}	0
							WrBk	P2	A1	20	A1	Unca.	{}	20
				Excl.	A2	40	DaRp	P2	A2	0	A2	Excl.	{P2}	0

•Assumes A1 and A2 *map* to same cache location but are not in the same memory block (so not in the same cache block)

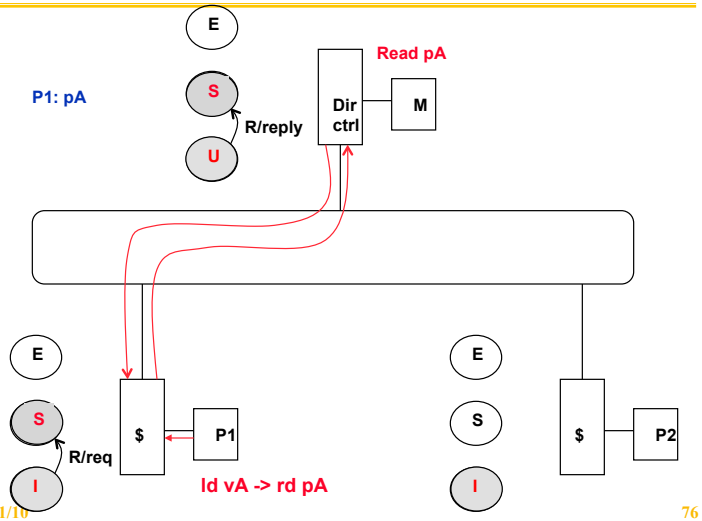
* Initial cache state is invalid

* Assume write allocate

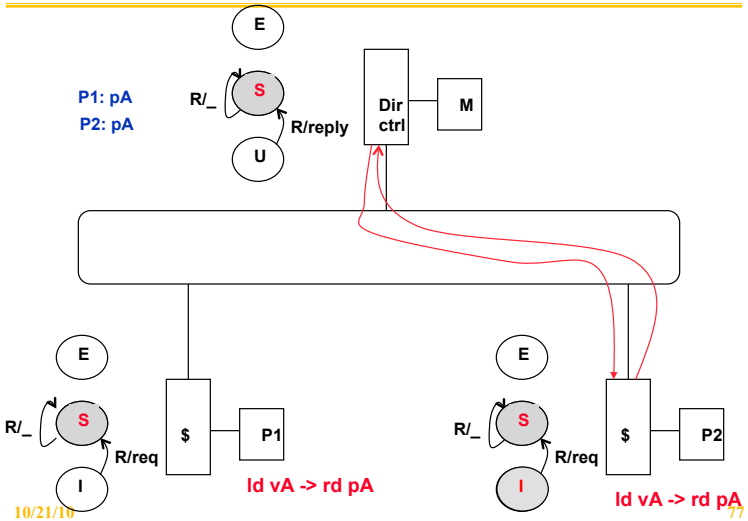
Implementing a Directory

- **We assume operations atomic, but they are not; reality is much harder; must avoid deadlock when run out of bufffers in network (see Appendix E)**
- **Optimizations:**
  - **read miss or write miss in Exclusive: send data directly to requestor from owner vs. 1st to memory and then from memory to requestor**

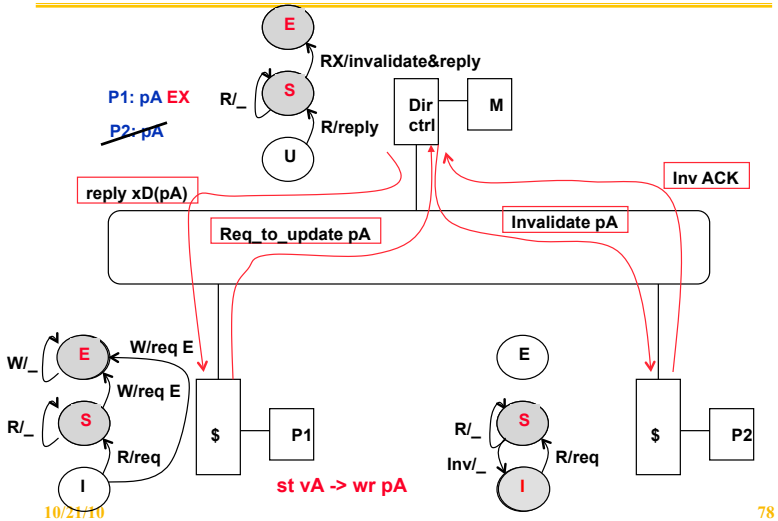
Example Directory Protocol (1st Read)



Example Directory Protocol (Read Share)



Example Directory Protocol (Wr to shared)



Example Directory Protocol (Wr to Ex)